XML pull parsing patterns

By Aleksander Slominski
Version $Id: patterns.html,v 1.4 2002/08/16 16:23:02 aslom Exp $

As XML pull parsing is gaining acceptance it is important to describe best practices. The aim is to make it easier to write code that process XML input with XML pull parser by encapsulating best current practices.

XML pull parsing is an alternative to push parsing approach that is very well suited for processing every element of XML document in a streaming fashion. In contrast XML push parsing APIs such as SAX are especially good to process only parts (filtering) of XML input as they use callback model (reactor pattern). In SAX user code is called by parser when interesting part of XML input is available and it is user code responsibility to keep state between callbacks. That is main difference when comparing to pull parsing: when using XML pull parser the user code is in control and can pull more data when it is ready to process it. However those two models should not be regarded as competitors but as complimentary approaches to XML parsing. It is possible to have API that will allow users can easily switch between pull and push parsing even when working on the same input source (details of such approach is described in more detail in my technical report).

As XML pull parsing is not in widespread usage it is useful to see what are common usage patterns that stems from my own experience of using XML pull parsing APIs for last two years.

Known patterns:

- parsing code mirrors XML structure
- how to safely and efficiently parse XML content that is sequence of known elements
- how to parse element content that may be in any order and dispatch actions based on element content
- how to skip unknown child content
- more patters soon!

All sample code is using XmlPull V1 API and is available in file MyAddressBook.java.

Additional utility functions (such as implementation of SKIP pattern are available at org\xmlpull\v1\util\XmlPullWrapper.java.

MIRROR: parsing code mirrors XML structure

Intent

When parsing nested data structures it is good idea to have code written in such way to reflect structure of XML document.

Motivation

It is easy to do with XML pull parsing - one can easily walk through input and delegate more difficult parsing to specialized methods that encapsulates required knowledge about parsing of well determined XML fragment.

It is convenient and very natural to look on XML as tree. Therefore if program will also mirror this XML structure it will be easy to understand and to maintain.

Applicability

in Model-View-Controller patterns (MVC) use it to create model, XML data binding etc.

Sample code

Consider application that is using XML to store persona; address book. Each entry in address book describes one person and each person may contain name and address. When application reads XML it will recreate address book content in memory.

XML input	Program representation
<**addressbook** xmlns="http://tempuri.org/addressbook"> <**person**> <**name**>Joe Doe</name> <address>Sesame Street</address> </person> <**person**> <**name**>Joe Doe 2</name> </person> </addressbook>	**ADDRESSBOOK** object or procedure **PERSON**: entry in ADDRESSBOOK **NAME**: field in PERSON **ADDRESS**: field in PERSON **PERSON**: entry in ADDRESSBOOK **NAME**: field in PERSON

SEQUENCE: parsing sequence of similar elements in loop

Intent

Simplify processing XML content that is organized in sequence.

Motivation

This is very common that XML is used to represent potentially unlimited number of entries. This is very common situation when pull parser is used to process incoming streamed XML data of known structure. Using this pattern it is possible to process large XML input in very limited memory as need for any in-memory XML representation is minimal.

Applicability

databases, array like structures encoding (ex. array in SOAP encoding)

Implementation

Typical sequence will consist of similar items. When starting processing parser is positioned on start element of list.

Then processing is done in loop that checks each child element until it reaches END_TAG for list itself.

```
move parser to list START_TAG
while( has child ) {
  if (child is known tag) {
     process it
  } else {
```

```
    } throw exception or skip unknown child content see SKIP pattern
    }
  }
```

more specifically in case of XmlPull API:

```
  parser.require( pp.START_TAG, PARENT_NAMESPACE, PARENT_NAME);
  whil ( parser.nextTag() == pp.START_TAG ) {
    if( CHILD_NAME.equals(parser.getName()) && CHILD_NAMESPACE.equals(parser.getNamespace()) ){
      processSequenceElement (parser)
    } else {
      //throw exception or skip unknown child content see SKIP pattern
      wrapper.skipSubTree();
    }
  }
  parser.require( pp.END_TAG, PARENT_NAMESPACE, PARENT_NAME);
```

and this can be used as a template to implement this pattern.

Sample code

Let consider this simple XML code that represents address book consisting of list of persons.

```
<addressbook xmlns="http://tempuri.org/addressbook">
  <person>
    <name>Joe Doe</name>
    <address>Sesame Street</address>
  </person>
  <person>
    <name>Joe Doe 2</name>
  </person>
</addressbook>
```

This XML naturally translates to two function - first we need need to get each address book entry (readAddressBook) and then to parse each entry (readPerson). As a result of calling first function user will get at Vector with list of entries that were in XML:

```
public Vector readAddressBook (XmlPullParser pp) throws XmlPullParserException, IOException
 {
       Vector addressBook = new Vector();
```

State of input is represented by XML pull parser instance (passed as pp parameter to this function). We need to move parser to first start tag, check that it has name "addressbook" and namespace http://tempuri.org/addressbook

```
       pp.nextTag();
       pp.require(pp.START_TAG, SAMPLE_NS, "addressbook");
```

Now we need to loop through every start tag, extract person info and keep looping until "addressbook" end tag is reached

```
       // read each person - using nextTag() guarantees that only START_TAG or END_TAG may be r
       while( pp.nextTag()) == pp.START_TAG) {
```

parsing of <person> start tag is delegated to another method - that makes code nicer to read. notice also that as nextTag can only return START_TAG or END_TAG so at this point we know that parser is on START_TAG so

calling readPerson() is safe (it expect parser to be positioned on <peron> start tag):

```
            Person person = readPerson(pp);
            addressBook.add( person );
        }
```

Notice that we not need to check if parser is on "addreessbook" end tag - at this point we know that it is at end tag and if readPerson() worked correctly this must be end tag

```
        return addressBook;
    }
```

Similarly another function (readPerson) will extract info from XML to create Person object (see also next pattern).

For details see method readAddressBook() sample code in MyAddressBook.java.

Additional background information

for more information about how pull parsing API compares ot push APIs (such as SAX) see section "Push and Pull: complementary sides of XML parsing" in technical report "Design of a Pull and Push Parser System for Streaming XML" [http://www.extreme.indiana.edu/xgws/papers/xml_push_pull/]

DISPATCH: parsing element content that has elements in any order

Intent

When processing element content that can have children in any order.

Motivation

When element can contain the element in **any order** it looks like it may be more difficult to parse (and validate input). However using this pattern the task is greatly simplified. Essentially one needs to keep parsing element content in loop and keeping enough state in loop to be able to determine that element content is valid.

Applicability

in Model-View-Controller patterns (MVC) use it to create model, XML data binding etc.

Implementation

Typical container element will have children elements in any order.

Processing is done in loop that checks each child element and dispatches processing to appropriate handler method. In this regard this pattern is very similar to SEQUENCE however children of each kind will be processed at most once:

```
move parser to container START_TAG
while( has child ) {
  if (child is known tag) {
     if(already processed child) throw exception
     process child element
```

```
    } set flag that this child was processed
    } else if (child is another known tag) {
       if(already processed child) throw exception
       process child element
       set flag that this child was processed
    } else {
       throw exception or skip unknown child content see SKIP pattern
    }
  }
```

more specifically in case of XmlPull API:

```
parser.require( pp.START_TAG, PARENT_NAMESPACE, PARENT_NAME);
while( parser.nextTag() == pp.START_TAG ) {
  if( CHILD_NAME.equals(parser.getName()) && CHILD_NAMESPACE.equals(parser.getNamespace()) )
     if(already proceed child) throw exception
     processChild (parser)
  } else if (ANOTHER_CHILD_NAME.equals(parser.getName()) && ANOTHER_CHILD_NAMESPACE.equals(pa
     if(already proceed child) throw exception
     processAnotherChild (parser)
  } else {
     //throw exception or skip unknown child content see SKIP pattern
     wrapper.skipSubTree();
  }
}
parser.require( pp.END_TAG, PARENT_NAMESPACE, PARENT_NAME);
```

and this can be used as a template to implement this pattern.

Sample code

We will refer to the same example of address book (see above). However here we have situation in which person content can have following content:

- <name> element followed by <address> element
- <address> element followed by <name> element
- only <name> element
- only <address> element
- no <name> and no <address>

and it can get easily more complex - we need here for example detect invalid content:

- element that name is not "address" or "name"
- duplicated "name" or "address" elements

Here is an example code that parser <person> element and conforms to those conditions:

```
public Person readPerson (XmlPullParser pp) throws XmlPullParserException, IOException
{
```

it is always safe to do some extra checking of assertions:

```
  pp.require(pp.START_TAG, SAMPLE_NS, "person");
  // read person fields
  Person person = new Person();
```

this loop will be exit when end tag for </person> is encountered:

```
  while( pp.nextTag() == pp.START_TAG) {
```

as loop condition called nextTag() so the only possible states here are START_TAG and END_TAG however loop condition eliminates END_TAG so content MUST be START_TAG (therefore check "`if(eventType == pp.START_TAG)`" would be redundant!).

At this point we can have any element content of person element in any order so we use if-else-if construction to check for allowed element content. Extra state needs to be preserved here t be able to detect if content is invalid (in this example case when name element is duplicated - see below)

```
      if(pp.getName().equals("name")) {
```

here is one of constraints checked - person name should not be duplicated:

```
        if(person.name != null) {
           throw new RuntimeException("person can only have one name"
                     +pp.getPositionDescription());
         }
```

now we can retrieve person name - note that nextText() will move parser to name end tag - precisely where we need it!

```
         person.name = pp.nextText();
```

this is the same case as above for name element but here we read address element:

```
      } else if(pp.getName().equals("address")) {
         if(person.address != null) {
           throw new RuntimeException("person can only have one address"
                     +pp.getPositionDescription());
         }
         person.address= pp.nextText();
```

if we are at this place we have start tag but it is not recognized: either ignore or throw error!

```
      } else {
        //throw new RuntimeException("unknow person tag "+pp.getName());
        wrapper.skipSubTree();
      }
  }
  return person;
}
```

for details see method readPerson() sample code in MyAddressBook.java.

SKIP: parsing element content that has elements in any order

Intent

When processing element content that may be extended in future it is wise to skip unknown element sub trees.

Motivation

This allows for more robust XML handling code.

Applicability

in situations when XML can be extended later

Sample code

We will refer to the same example of address book (see above). However we will modify code to actually ignore unknown child elements:

```
} else {
  wrapper.skipSubTree();
}
```

where skipSubTree is very simple function that skips all child content (can be downloaded from org\xmlpull\v1\util\XmlPullWrapper.java) :

```
public void skipSubTree()
    throws XmlPullParserException, IOException
{
     pp.require(pp.START_TAG, null, null);
     int level = 1;
     while(level > 0) {
         int eventType = pp.next();
         if(eventType == pp.END_TAG) {
             --level;
         } else if(eventType == pp.START_TAG) {
             ++level;
         }
     }
}
```

see MyAddressBook.java for more detailed example

STILL UNDER DEVELOPMENT ...

BLOCKS: Combine pull operations into blocks

This one of main advantages of pull parsing is keeping parsing control in user code. That means that it is easily to combine multiple parsing operations into useful and reusable blocks. Those blocks can be quite low level such as nextText() that combines next(), getEventType() and getText() into handy function to extract text from element or they can be higher level procedures such as described in two previous patterns. It is also wise to use and reuse such blocks as they are not only good for modularizing XML parsing but also

The document is available at http://www.extreme.indiana.edu/~aslom/xmlpull/

Version: $Id: patterns.html,v 1.4 2002/08/16 16:23:02 aslom Exp $